Automatic test generation with ASMETA for the Mechanical Ventilator Milano controller

Andrea Bombarda$^{1\,[0000-0003-4244-9319]}$, Silvia Bonfanti$^{1\,[0000-0001-9679-4551]}$, and Angelo Gargantini$^{1\,[0000-0002-4035-0131]}$

Dipartimento di Ingegneria Gestionale, dell'Informazione e della Produzione,
Università degli Studi di Bergamo, Bergamo, Italy
{andrea.bombarda,silvia.bonfanti,angelo.gargantini}@unibg.it

Abstract. This paper presents an automatic test cases generation method from Abstract State Machine specifications. Starting from the ASMETA specification, the proposed approach applies the following steps: 1. Generation of abstract tests from a ASMETA model; 2. Optimization of the abstract tests; 3. Concretization of the abstract tests in GoogleTest; 4. Execution of the concrete tests on C++ code. We have applied this approach to the Mechanical Ventilator Milano (MVM) project, which our research group has contributed to develop, test, and certify during the Covid-19 pandemic.

1 Introduction

In response to the lack of ventilators due to Covid-19, a group of physicists, engineers, physicians, computer scientists, and others from 12 countries around the world has developed a simplified mechanical lung ventilator, called MVM (Mechanical Ventilator Milano)[1]. The project started from an idea of the physicist Cristiano Galbiati, who was also the leader, and our research group has been involved in the development and testing of the device, in order to get the certifications from local authorities and distribute the MVM in the hospitals of different countries. In only 42 days from the initial prototype production to the demonstration of performances, the FDA (Food and Drug Administration) declared that the MVM falls within the scope of the Emergency Use Authorization (EUA) for ventilators and, during the following months, it has obtained the Health Canada and the CE marking as well. Thanks to these achievements, the MVM can be sold and used in the USA, but also in Canada and Europe.

During the development, as required by the standards, we (together with other colleagues) started to design the MVM controller (more details can be found in [1]) and we have used the Yakindu Statechart Tool. Regarding unit testing, since Yakindu does not provide an automatic test generator, tests were manually written and we were able to test the entire machine in a satisfactory way, enough to obtain the required certifications. As well known, writing tests manually should be discouraged, especially if a model is present, since it requires a significant amount of time and can be an error-prone activity. Therefore, after the completion of the development and certification process, we have wondered if test generation starting from formal specifications would have

[1] https://mvm.care/

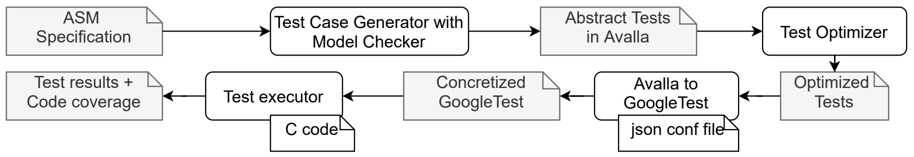

Fig. 1: Test generation and execution process

been applicable in this case and since we still have access to the source code of the MVM, we decided to apply a Model-based-testing (MBT) method to this project. We have decided to use the ASMETA [2] framework which we are familiar with and which offers all the necessary techniques (including those for V&V, missing in Yakindu).

We have started from the ASM specifications of the MVM controller and we have validated and verified our formal specification. Using a model checker we have generated abstract test sequences, that have been optimized and, then, concretized in order to be executable on the C++ implementation of the MVM controller. By evaluating the coverage reached with this testing process, we have obtained better results than the one got with manual tests. The entire process is presented in the following sections.

2 Test generation

The testing process applied to the case study is depicted in Fig. 1. It starts from the ASM specifications, which have been validated, and verified - but the modeling activity is not reported here for brevity. Starting from the ASM model, abstract tests are generated by the `ATGT` tool, exploiting the counterexample generation of the model checker. Tests are saved in Avalla, the language used to write scenarios in ASMETA [2]. In this paper, we extend the approach presented in [11] by generating test sequences using the bounded model checker (BMC) and Linear Temporal Logic (LTL) properties. `ATGT` generates the test predicates which are then translated to suitable LTL temporal properties, called *trap properties* whose counterexamples generated by the BMC are the tests we are looking for. Test predicates are generated by applying the following coverage criteria: 1. *Basic rule* (every rule r_i is executed at least once), 2. *Complete rule* (every rule is executed and performs a non trivial update), 3. *Rule update* (every update is performed once and it is not trivial), 4. *Rule guard* (every guard is evaluated true at least once, and false at least once), 5. *MCDC* (Modified Condition Decision coverage of the guards), 6. *2-wise* (pairwise testing of all the inputs - with a limited domain).

In this case study, tests are generated using the *monitoring* optimization: when a test sequence *ts* is generated for a test predicate yet to cover, the algorithms checks if *ts* covers accidentally other test predicates and it skips the test predicates already covered.

Moreover, we have introduced a *timeout*: for every test predicate *tp* to be covered, the model checker is interrupted if it reaches the timeout before producing a test, either because the test that covers *tp* exists but the model checker is unable to find it or because *tp* is unfeasible, i.e. there is no test that covers it and the trap property is actually true. However, because ATGT uses the classical bounded model checking, it is unable to distinguish the two cases by proving test predicates unfeasibility.

Table 1: Comparison between different criteria for automatic test cases generation

Criteria	#Tps	Timeout 10 minutes #Tests	#Time-outs	Generation time [min]	#Tps covered	%Tps covered	Timeout 40 minutes #Tests	#Time-outs	Generation time [min]	#Tps covered	%Tps covered
Basic rule	72	13	29	345	43	60%	24	11	773	61	85%
Complete rule	2	0	0	0	2	100%	0	0	0	2	100%
Rule guard	124	1	60	601	64	52%	1	27	1080	97	78%
Rule update	89	0	52	520	37	42%	0	25	1000	64	72%
MCDC	148	10	55	581	93	63%	9	24	997	124	84%
2-wise	420	77	0	1	420	100%	73	0	1	420	100%
All criteria	853	101	196	2048	659	77%	107	87	3852	768	90%

Code 1: Original scenario

```
scenario
check apneaAlarm = false;
check iValve = CLOSED;  ...
set respirationMode := PCV;  ...
step
check apneaAlarm = false;
check iValve = CLOSED;  ...
set respirationMode := PCV;  ...
step
check iValve = OPEN;  ...
```

Code 2: Check Opt.

```
scenario
check apneaAlarm = false;
check iValve = CLOSED;  ...
set respirationMode := PCV;  ...
step  ...
set respirationMode := PCV;  ...
step
check iValve = OPEN;  ...
```

Code 3: Set Opt.

```
scenario
check apneaAlarm = false;
check iValve = CLOSED;  ...
set respirationMode := PCV;  ...
step  ...
step
check iValve = OPEN;
```

Tab. 1 reports the comparison in terms of test predicates, number of generated tests, number of timeouts, generation time, and number of test predicates covered for each coverage criteria by setting two different timeouts of 10 and 40 minutes.

We can observe that by increasing the timeout, the total number of covered test predicates increases (from 77% with timeout 10 minutes to 90% with timeout 40 minutes) as well as the number of generated tests and the generation time. For each coverage criterion, the number of *tps* covered increases from 10 minutes timeout to 40 minutes timeout, except for *complete rule* and *2-wise* criteria. Considering *complete rule*, no test is generated because, due to the *monitoring* optimization, all *tps* are already covered by tests generated with *basic rule* criteria. For the *2-wise* criterion, in both cases all the *tps* are covered. There is a difference in the number of generated tests that is greater with 10 minutes timeout. This is because of the lower timeout, fewer tests are generated before trying to cover *2-wise*, so more *tps* will result uncovered and they will need more tests.

Remark: We suspect that many of the uncovered test predicates are unfeasible. Being able to prove unfeasible test requirements is necessary to give complete information about the real coverage of the specification. We plan to extend ATGT in order to support proof by induction using the IC3 algorithm (as we did for property verification).

Test optimization. Once the abstract tests are generated, we perform the following (optional) test optimizations, which do not change the semantic of the tests but improve the readability and the translatability of the abstract tests to concrete ones.
1. Check optimization: This operation removes unchanged controlled locations. If a controlled location in state s_i has not changed w.r.t. state s_{i-1}, the corresponding **check** is removed, if present. For instance, in Code 1 **check** command on the location *iValve* is repeated, so it is possible to remove the second one (see Code 2).

```
[{”asmName”: ”startVentilation”,
   ”cName”: ”startVentilation”,
   ”commandType”: ”IN_EVENT”
},{”asmName”: ”time”, ”cName”: ”time”,
   ”commandType”: ”TBD”
},{”asmName”: ”iValve”,
   ”cName”: ”defaultMock−>getInValveStatus”,
   ”commandType”: ”OPERATION”
},{”asmName”: ”state”, ”cName”: ”state”,
   ”commandType”: ”STATE”
},{”asmName”: ”mode”, ”cName”: ”mode”,
   ”commandType”: ”VAR”}]
```

Code 4: JSON file for function mapping

2. Set optimization: It aims to remove **set** commands of monitored variables in state s_{i-1} if they are not actually asked to compute the update set for state s_i. In Code 2, the second instance of set respirationMode = PCV is removed in Code 3 since it is useless.

The average number of check and set per state in the generated scenarios without optimization is respectively 37 and 15. By applying the check optimization technique the optimized scenarios have an average of 11.22 check per state, while by applying the set optimization the average of set per state became 3.31.

3 Test concretization

We have concretized Avalla tests as unit tests in the GoogleTest framework, in order to be executed on the C++ code of the MVM controller generated by Yakindu. The concretization process consists of the following three consecutive steps explained below.

1. Mapping of ASMETA functions to state machine variables. To map each ASM function with the corresponding function in C++ code, we introduce a configuration file in the JSON format. It is automatically generated and filled with all the functions set or checked in the Avalla scenarios which can be adjusted manually.

For each function, the JSON file contains: • asmName, i.e., the name of the function in the ASMETA model; • cName, representing the name of the corresponding function in C++ code; • commandType indicating the type of the function chosen between IN_EVENT, VAR, OPERATION, STATE, and TBD (TBD is the default type and TBD functions are ignored during test concretization).

An example of the JSON file is reported in Code 4. The functions startVentilation is IN_EVENT since it is raised by the user. mode is VAR because it represents an internal field of the state machine, and iValve is OPERATION function type because it interacts with hardware components, i.e. the input valve. Functions used only in the ASMETA model but not in the C++ code (e.g. time) are set to be ignored (TBD type).

2. Hardware mocking: Since the MVM state machine interacts with hardware, during test concretization we needed to append in the generated C++ test file some hardware mock. It has been written using the same interface as the real classes of the hardware components and it is included automatically by the scenario concretization process.

3. GoogleTest code generation: Starting from the Avalla scenarios and using the JSON configuration and the mocking files, we have concretized the tests in GoogleTest. MVM has been developed as a cycle-based state machine. For this reason, we have defined the main cycle duration of 100 ms as in the C++ implementation, which is used to convert the step command in proceed_time(100) command in GoogleTest. The other Avalla

```
set mode := PSV;
set startVentilation := true;
step
check time = 3;
check oValve = CLOSED;
check iValve = OPEN;
check state =
        MAIN_REGION_PSV_R1_INSPIRATION;
```

```
sm−>setMode(PSV);
sm−>raiseStartVentilation();
runner−>proceed_time(100);

EXPECT_EQ(valveMock−>getOutValveStatus() , CLOSED);
EXPECT_EQ(valveMock−>getInValveStatus() , OPEN);
EXPECT_TRUE(sm−>isStateActive(
        MAIN_REGION_PSV_R1_INSPIRATION));
```

Code 5: Test concretization from an Avalla scenario fragment to a GoogleTest test case

commands are concretized as explained in Tab. 2. Code 5 shows a test concretization example of an Avalla scenario. IN_EVENT functions (such as startVentilation) are raised only when they are set to true in the Avalla scenario. VAR functions, like mode, are set in the GoogleTest test case when there is a corresponding set in the Avalla scenario, while OPERATION functions, such as iValve, are translated in method calls. VAR and OPERATION functions are controlled when they are checked in the Avalla scenario. Finally, the STATE function represents the active state of the machine.

4 Test execution

Having concretized the optimized tests, we have tested the C++ code of the MVM controller with them. Tab. 3 reports the incremental coverage reached with the tests. These results confirm those reported in Tab. 1: increasing the timeout leads to an increment of the covered test predicates and the code. The table shows that every criterion, except *complete rule* and *rule update* for which no test is generated, improves the code coverage, so we cannot claim that any criterion could have been skipped.

We believe that higher values of coverage are difficult to be obtained since we started from the code generated by Yakindu SCT and many parts of the code are only used by Yakindu itself and can not be mapped in external calls.

Remark: Generating code automatically may hinder testers in reaching a complete code coverage because some code could never be covered or because it would require adding ad hoc tests that can not be easily derived from the behavior specifications.

With automatic test case generation, we are able to improve the coverage of the controller compared to the coverage obtained with handwritten test cases. Nevertheless unit testing the MVM controller was not mandatory in order to obtain the safety certification, it is important, since its behavior affects the valves position. However, these tests can be used for integration testing, which is mandatory for the certification.

Table 2: Translation rules between Avalla and GoogleTest instructions (sm is the generic name used to indicate the state machine object in Yakindu)

Function type	Set	Check
STATE	`//`	`EXPECT_TRUE(sm->isStateActive([stateName]))`
IN_EVENT	`sm->raise[cName]()`	`//`
VAR	`sm->set[cName]([value])`	`EXPECT_EQ(sm->get[cName](),[value])`
OPERATION	`[cName]([value])`	`EXPECT_EQ([cName](),[value])`

Table 3: Coverage reached using different timeouts and coverage criteria

Criteria	Timeout 10 minutes			Timeout 40 minutes		
	Statement Cov.	Branch Cov.	Function Cov.	Statement Cov.	Branch Cov.	Function Cov.
Basic rule	65.69%	63.48%	59.46%	80.97%	81.32%	79.05%
Complete rule	65.69%	63.48%	59.46%	80.97%	81.32%	79.05%
Rule guard	66.19%	63.91%	60.47%	81.48%	81.74%	80.07%
Rule update	66.19%	63.91%	60.47%	81.48%	81.74%	80.07%
MCDC	70.24%	69.85%	65.54%	81.48%	81.74%	80.07%
2-wise	70.24%	69.85%	65.54%	81.98%	82.17%	81.08%
All criteria	70.24%	69.85%	65.54%	81.98%	82.17%	81.08%

5 Related works

In this paper, we generate concrete tests starting from the formal specification of the MVM controller. In [5, 6], from an ASMETA specification, C++ unit tests are automatically generated (using the Boost Test library) and they are executed against the C++ code automatically generated from the ASMETA specification. The main difference with the approach presented in this paper is that the code is already available and a map between ASMETA functions and C++ functions is required. This is a widely used methodology, especially when the formal model of the SUT is available [10, 12]. In this work and in [4], we start from the ASM specification of the SUT, but many other techniques have been used in the literature. For example, in [3] tests are generated off-line, starting from a Timed Output Input Symbolic Transition System. This process is known as *conformance testing* since the testers want to check the conformance between formal specifications and the actual system [8]. FSMs, or their extensions, are often used for this purpose [9, 13, 14]. However, the concretization of the generated tests has to be performed in different ways in order to be executed against the actual implementation. In this paper, we propose a test concretization methodology starting from Avalla scenarios and resulting in a collection of GoogleTest test cases. Other approaches exploit different tools, such as the ACT one [7] that can be used for concretizing abstract tests from formal specifications of web applications. Though Yakindu does not have an integrated model checker, other tools like Gamma [15] provide an environment to verify properties but they do not generate automatically an entire test suite.

6 Conclusion

In this paper, we have presented an approach to automatically generate test cases from ASMETA specification. The ATGT tool generates abstract tests by means of the model checker and then they are concretized into GogoleTest cases. The unit tests are then executed on C++ code and test results and code coverage are collected. This approach has been successfully applied to the MVM case study, the code coverage is increased compared to the one obtained with handwritten tests. As future work, we plan to compare probabilistic random test generation instead of using the model checker, since the counterexample generation is very time-consuming.

References

1. Abba, A., et al.: The novel mechanical ventilator milano for the COVID-19 pandemic. Physics of Fluids **33**(3), 037122 (mar 2021). https://doi.org/10.1063/5.0044445
2. Arcaini, P., Bombarda, A., Bonfanti, S., Gargantini, A., Riccobene, E., Scandurra, P.: The ASMETA Approach to Safety Assurance of Software Systems, pp. 215–238. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-76020-5_13
3. Bannour, B., Escobedo, J.P., Gaston, C., Le Gall, P.: Off-line test case generation for timed symbolic model-based conformance testing. In: Nielsen, B., Weise, C. (eds.) Testing Software and Systems. pp. 119–135. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
4. Bombarda, A., Bonfanti, S., Gargantini, A., Radavelli, M., Duan, F., Lei, Y.: Combining model refinement and test generation for conformance testing of the IEEE PHD protocol using abstract state machines. In: Testing Software and Systems, pp. 67–85. Springer International Publishing (2019)
5. Bonfanti, S., Gargantini, A., Mashkoor, A.: Generation of C++ unit tests from abstract state machines specifications. In: 2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). IEEE (apr 2018). https://doi.org/10.1109/icstw.2018.00049
6. Bonfanti, S., Gargantini, A., Mashkoor, A.: Design and validation of a C++ code generator from Abstract State Machines specifications. Journal of Software: Evolution and Process **32**(2) (nov 2019)
7. Bubna, K., Chakrabarti, S.: Act (abstract to concrete tests) - a tool for generating concrete test cases from formal specification of web applications. In: ModSym+SAAAS@ISEC (2016)
8. Cavalli, A.R., Maigron, P., Kim, S.U.: Automated protocol conformance test generation based on formal methods for lotos specifications. In: Proceedings of the IFIP TC6/WG6.1 Fifth International Workshop on Protocol Test Systems V. p. 237–248. North-Holland Publishing Co., NLD (1992)
9. Dorofeeva, R., El-Fakih, K., Maag, S., Cavalli, A.R., Yevtushenko, N.: FSM-based conformance testing methods: A survey annotated with experimental evaluation. Information and Software Technology **52**(12), 1286–1297 (dec 2010). https://doi.org/10.1016/j.infsof.2010.07.001
10. Fraser, G., Wotawa, F., Ammann, P.E.: Testing with model checkers: a survey. Software Testing, Verification and Reliability **19**(3), 215–261 (sep 2009). https://doi.org/10.1002/stvr.402
11. Gargantini, A., Riccobene, E.: ASM-based testing: Coverage criteria and automatic test sequence generation. JUCS - Journal of Universal Computer Science **7**(11), 1050–1067 (nov 2001). https://doi.org/10.3217/jucs-007-11-1050
12. Hong, H., Lee, I., Sokolsky, O.: Automatic test generation from statecharts using model checking. Technical Reports (CIS) (10 2001)
13. Kalaji, A., Hierons, R.M., Swift, S.: A search-based approach for automatic test generation from extended finite state machine (EFSM). In: 2009 Testing: Academic and Industrial Conference - Practice and Research Techniques. pp. 131–132 (2009). https://doi.org/10.1109/TAICPART.2009.19
14. Merayo, M.G., Núñez, M., Rodríguez, I.: Formal testing from timed finite state machines. Computer Networks **52**(2), 432–460 (feb 2008). https://doi.org/10.1016/j.comnet.2007.10.002
15. Molnár, V., Graics, B., Vörös, A., Majzik, I., Varró, D.: The Gamma statechart composition framework: design, verification and code generation for component-based reactive systems. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings. pp. 113–116. ACM (2018). https://doi.org/10.1145/3183440.3183489